Automatic Benchmark Profiling through Advanced Trace Analysis

Alexis Martin, Vania Marangozova-Martin

Automatic Benchmark Profiling through Advanced Trace Analysis

Alexis Martin , Vania Marangozova-Martin

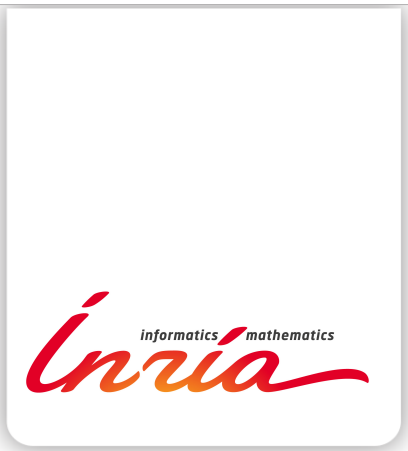
RESEARCH REPORT

N° 8889

March 23, 2016

Project-Team Polaris

ISSN 0249-6399 ISRN INRIA/RR--8889--FR+ENG



Automatic Benchmark Profiling through Advanced Trace Analysis

Alexis Martin * † ‡, Vania Marangozova-Martin * † ‡

Project-Team Polaris

Research Report n° 8889 — March 23, 2016 — 15 pages

Abstract: Benchmarking has proven to be crucial for the investigation of the behavior and performances of a system. However, the choice of relevant benchmarks still remains a challenge. To help the process of comparing and choosing among benchmarks, we propose a solution for automatic benchmark profiling. It computes unified benchmark profiles reflecting benchmarks' duration, function repartition, stability, CPU efficiency, parallelization and memory usage. It identifies the needed system information for profile computation, collects it from execution traces and produces profiles through efficient and reproducible trace analysis treatments. The paper presents the design, implementation and the evaluation of the approach.

Key-words: Benchmark, performance, analysis, trace, workflow, profile

This work is a part of the SoC-Trace FUI project.

* CNRS, LIG, F-38000 Grenoble, France
† Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
‡ Inria

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Profilage Automatique de Benchmark par Analyse Avancée de Traces d'Execution

Résumé : L'utilisation de programmes de benchmark est cruciale afin d'investiguer et de comprendre le comportement et les performances d'un système. Cependant, le choix du benchmark le plus pertinent reste un challenge. Nous proposons une méthode pour la génération de profils automatique de benchmarks, afin d'aider à la comparaison de ceux ci, et donc à la sélection d'un ou plusieurs benchmarks en particulier. Cette méthode permet de comparer les caractéristiques des benchmarks comme leur durée, la répartition de leur fonctions, la stabilité du benchmark, l'utilisation des processeurs, l'aspect parallèle et l'utilisation de la mémoire. Ces profiles sont générés à partir de traces d'execution et calculés en utilisant des traitements reproductibles et efficaces. Ce rapport expose le design, l'implémentation et l'évaluation de cette approche.

Mots-clés : Benchmark, performance, analyse, trace, workflow, profile

Contents

1 Introduction

System performance is a major preoccupation during system design and implementation. Even if some performance aspects may be guaranteed by design using formal methods [1], all systems undergo a testing phase during which their execution is evaluated. The evaluation typically consists in quantifying performance metrics or in checking behavior correction in a set of use cases. In many cases, system performance evaluation does not only consider absolute measures for performance metrics but is completed by *benchmarks*. The point is to use well-known and accepted test programs to compare the target system against competitor solutions.

Existing benchmarks are numerous and target different application domains, different platforms and different types of evaluation. There are benchmarks for MPI applications [2, 3], for mobile devices [4], for web servers [5, 6], for the Linux operating system [7, 8], etc.

Constructing a benchmark is a difficult task [9] as it needs to capture relevant system behaviors, under realistic workloads and provide interesting performance metrics. This is why benchmarks evolve with the maturation of a given application domain and new benchmarks appear as new system features need to be put forward. Developers frequently find themselves confronted with the challenge of choosing *the right* benchmark among the numerous available. To do so, they need to understand under which conditions the benchmark is applicable, what system characteristics it tests, how its different parameters should be configured and how to interpret the results. In most cases, the choice naturally goes to the most popular benchmarks. Unfortunately, these are not suitable for all use cases and an incorrect usage may lead to irrelevant results.

In this paper, we present our solution for automatic profiling of benchmarks. The profiles characterize the runtime behavior of benchmarks using well defined metrics and thus help benchmark comparison and developers' choices. The profile computation uses information contained in execution traces and is structured as a deterministic trace analysis workflow. The contributions of the paper can be summarized as follows:

- *Definition of unified profiles for benchmarks.* We define profiles in terms of execution duration, function repartition, stability, CPU efficiency, parallelization and memory usage. These are standard metrics and can be easily understood and interpreted by developers.

- *Definition of the tools needed to compute the profiles.* We structure the computation as a reproducible workflow and stress the importance of parallel and streaming features.

- *Definition of the data needed for profile computation.* We use system tracing and extract useful data in application-agnostic manner.

- *Validation of our solution.* We validate our solution with the Phoronix Test Suite [8] run on different embedded and desktop platforms.

The paper is organized as follows. Section 2 presents the design ideas behind our solution. It introduces the considered benchmark profiles and explains what is the process of their generation. Section 3 discusses our implementation by considering the Phoronix use case. This section shows how the benchmark profile information may be used to choose among benchmarks. Section 4 distinguishes our proposal from related work. Finally, Section 5 presents the conclusions and the perspectives of this work.

2 Automatic Profiling of Benchmarks

This section presents the benchmark profiles (Section 2.1), the needed data for their computation (Section 2.2) and the computation process itself (Section 2.3).

2.1 Benchmark Profiles Definition

The profile considered for a benchmark is independent of its semantics and is composed of the following features:

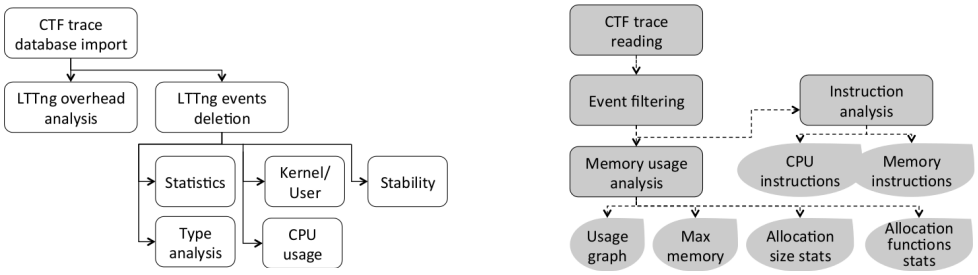
- *Duration.* This metric gives the time needed to run the benchmark. It allows developers to estimate the time-cost of a benchmarking process and to choose between short and long-running benchmarks.
- *CPU Occupation.* This metric characterizes the way a benchmark runs on the target system's available processors. It gives information about the CPU usage, as well as about the benchmark's parallelization. Indeed, it is interesting to know whether the benchmark is sequential or it may benefit from the presence of multiple processors.
- *Kernel vs User Time.* This metric gives the distribution of the benchmark execution time between the benchmark-specific (user) and kernel operations. This gives initial information on the parts of the system that are stressed by the benchmark.
- *Benchmark Type.* The type of a benchmark is defined by to the part of the system which is stressed during the benchmarking process. Namely, we distinguish between benchmarks that stress the processor (CPU-intensive), the memory (memory-intensive), the system, the disk, the network or the graphical devices. The motivation behind this classification is that it is application-agnostic and may be applied to all kinds of benchmarks.
- *Memory Usage.* This part of the profile provides information about the memory footprint of the benchmark, as well as the memory allocation variations.
- *Stability.* This metric reflects the execution determinism of a benchmark, namely the possible variations of the above metrics across multiple runs.

2.2 Initial Profile Data

The computation of the above metrics needs detailed data about the execution of a benchmark. It needs timing information both of the benchmark's global execution and of its fine-grained operations. It needs system information about the number and the scheduling of the benchmark's execution flows. It needs data distinguishing and quantifying the benchmark's different operations.

To collect this data, we decide to use system tracing and work with a historical log containing timestamped information about the different execution events. To ensure minimal system intrusion, we propose to use LTTng [10, 11, 12] which is a *de facto* standard for tracing Linux systems.

LTTng is capable of tracing both the kernel and the user-level operations. Using LTTng's predefined *tracepoints* for the Linux kernel, it is possible to trace context switches, interruptions,system calls, memory management and I/O. Using LTTng's profiling capacities and user-defined tracepoints, it is possible to quantify computations and data accesses, as well as gain insight into the benchmark semantic specifics.



(a) Kernel Trace Analysis (b) User Trace Analysis

Figure 1: Profile Computation Process

2.3 Profile Computation

The profile computation is a two-phase process which respectively analyzes the kernel and the user-level traces.

The analysis of the kernel trace follows a workflow implemented using the VisTrails tool [13]. It thus benefits from its reusability, efficiency and reproducibility features. Indeed, on one hand, the workflow may be reused for the analysis of different benchmarks. On the other hand, when used for multiple analyses of a given trace, it reuses intermediary results and computes the analysis in a deterministic way.

The kernel trace analysis workflow is given in Figure 1a. It follows the standard logic where traces are first captured and stored, and then analyzed offline. The workflow steps are the following.

The first (top) step of the workflow imports the kernel-level trace into a relational database. The database uses a generic trace format which allows to consider not only CTF (the LTTng's trace format) but also other formats [14]. During this step is computed the timing information about processes' activity and about the repartition between kernel and user time of the benchmark.

The intermediary level steps focus on LTTng tracing. The second step characterizes LTTng's tracing overhead in terms of number of traced events and execution slowdown. The third step filters out LTTng-related events so as to allow further analysis to focus only on the benchmark's performances. As the trace is already imported in the database, the computation is done via SQL requests.

The bottom level steps provide execution statistics in terms of number of execution events and execution duration, categorize the execution events to characterize the type of the benchmark, computes the repartition between kernel and user time, analyze the CPU occupation analysis and explore benchmark stability.

The analysis of the user-level trace is dataflow and stream-oriented as shown in Figure 1b. This approach is motivated by the fact that the database-oriented store-and-later-analyze approach does not scale well in the case of big execution traces. Indeed, execution traces may easily size up to several GB and their database import and subsequent analysis is costly in terms of storage and computation time.

The user-level trace analysis comprises several modules that process the trace in a pipeline. The first module reads directly the initial trace and transfers it to the events filtering module. The filtering module forwards to the subsequent modules only the information related to the benchmark processes. The memory usage analysis module provides information about the variations in the benchmark's allocated memory, about the size of the allocated chunks of memory

and about the frequency of usage of the memory allocation functions. The trace also provides information about the hardware counters which are used to quantify the user-level computations and the memory accesses.

3 Profiling the Phoronix Test Suite

This section details our benchmark profiling in the Phoronix Test Suite case.

3.1 The Phoronix Test Suite

The *Phoronix Test Suite* (*PTS*) [8] provides a set of benchmarks targeting different aspects of a system. *PTS* is available on multiple platforms including Linux, MacOS, Windows, Solaris and BSD.

PTS comes with some 200 open-source test programs. It includes hardware benchmarks typically testing battery consumption, disk performance, processor efficiency or memory consumption. It also targets diverse environments including OpenGL, Apache, compilers, games and many others.

PTS provides little information about benchmarks' logic and internals. Even if each benchmark is tagged as one of *Disk*, *Graphics*, *Memory*, *Network*, *Processor* and *System*, supposedly to indicate which system part is tested, there is no further information on how this tag has been decided or how exactly the benchmark tests this system part.

The repartition of the benchmarks is highly irregular. If we consider that *PTS* benchmarks having the same tag form a benchmark family, the *Network* family contains only one test, while the *Processor* family contains around 80 tests. If, in the first case, a developer has no choice, in the second case, he/she will need to know more about the benchmarks to choose the most relevant.

3.2 Tracing Phoronix with LTTng

To obtain the maximum information about the kernel activity of a benchmark, we enable all LTTng kernel tracepoints. We are thus able to collect information about scheduling decisions, process management (`exec`, `clone`, `switch` and `exit` function calls) and kernel usage (syscalls). Associated with each traced event is a hardware (CPU) and a software (PID) provenance context.

To analyze which parts of the target system are tested and thus deduce the type of a benchmark, we have analyzed the types of kernel events and mapped them to the *PTS* family tags. For example, the `hmm_page_aloc` and `mm_page_free` are clearly events related to *Memory*-related activity, while `power_cpu_idle` and `htimer_expire` are related to the *Processor*. Table 1 gives the mapping between events and system activity.

User-level tracing is highly dependent of the application to trace and *PTS* benchmarks' are highly heterogenous. To provide a generic tracing solution, we focus on the interface that is commonly used by all benchmarks, namely the standard C library (*libc*). Redefining the `LD_PRELOAD` environment variable and overloading the *libc* functions, it is easy to obtain the information about the memory management functions (`malloc`, `calloc`,`realloc` and `free`) needed for the computation of benchmarks' memory profiles.

Another aspect we are interested in is to characterize the user level behavior of a benchmark in terms of CPU or memory-related activity. To do so, we use the information provided by hardware performance counters. In particular we use the `Instruction` counter, which gives the total number of instructions executed. We also use the `L1-dcache-loads` and `L1-dcache-stores` counters that provide the total number of L1 cache reads and L1 cache writes. As all data

Family	Events
Processor	`timer_*; hrtimer_*; itimer_*; power_*;irq_*; softirq_*;`
Memory	`kemem_*; mm_*`
System	`workqueue_*; signal_*; sched_*; module_*;rpm_*; lttng_*; rcu_*;`
	`regulator_*; regmap_*;regcache_*; random_*; console_*;gpio_*;`
Graphics	`v4l2_*; snd_*;`
Disk	`scsi_*; jbd2_*; block_*;`
Network	`udp_*; rpc_*; sock_*; skb_*; net_*; netif_*;napi_*;`

Table 1: Tagging LTTng Kernel Tracepoints

access go through the L1 cache, the sum of those two values gives the total number of data related instructions. To get the number of computation related instructions, we use the difference `Instruction` − (`L1-dcache-stores` + `L1-dcache-stores`).

3.3 Experimental Setup

We have worked with 10 recommended *PTS* bechmarks, namely `compressgzip`, `ffmpeg`, `scimark2`, `stream`, `ramspeed`, `idle`, `phpbench`, `pybench`, `network-loopback` and `dbench`. This set contains three bencharks from the *Processor* family, three from the *System* family, two from *Memory*, one *Disk* and one *Network* benchmark. We ignored the *Graphics* family as its benchmarks were not in the recommended list.

Each benchmarks is run with its default options as defined by the *PTS* system except for the number of runs. Instead of 3 times we run benchmarks 32 times to ensure statistically reliable results [15]. The score for each benchmark, which is benchmark-specific, is computed as the mean value of the 32 obtained scores.

The experiments have been run on three different platforms which helped validate the fact that benchmarks have similar executions and hence have the same profile whatever the platform. We have used one UDOO board [16], one Juno board [17] and a desktop machine. The UDOO board has an i.MX 6 4core ARM CPU at 1GHz, a Cortex-M3 coprocessor and 1GB of RAM. It runs the multi-platform Debian kernel for ARM `armmp3.16`.The Juno board has one 2core CortexA57 and one 4core CortexA53 processors with 2GB of RAM. It runs a Debian kernel (`4.3.0-1-arm64`). The desktop platform has one quad-core Intel(R) Xeon(R) CPU E3-1225 v3 @3.20GHz with 32GB of RAM. It runs fedora with the 3.19 Linux kernel.

3.4 LTTng Overhead and Benchmark Stability

Our analysis starts with an evaluation of LTTng's perturbation of the target system. In terms of execution duration, both for kernel and user traces, LTTng's overhead is negligible as it is less that 1%. The only notable exception is the case of the `phpbench` benchmark slowed down by 78% by user-level tracing because of the its heavy use of memory operations.

In terms of collected events, LTTng-related events account for 10% to 26% in kernel traces and between $100K$ and $200K$ un user traces. To prevent bias in statistics metrics computations, these events are filtered out and ignored during trace analysis. Finally, considering benchmarks' results, scores from executions with and without tracing do not differ more than 2.5%.

Table 2 summarizes information about the 32 runs of the considered benchmarks. Namely we have the global execution time, the corresponding trace size, the relative time spent in idle mode (`idle` stands out), the standard deviation for benchmarks' duration and the ratio between user and kernel time. There are important differences, even between benchmarks belonging to

the same family. A simple recommendation to developers would be to use shorter benchmarks.

Benchmark	Exec.(m)	Size(GB)	Idle	SD	User / Kernel ratio
compress-gzip	94	5.18	76	0.03	78% 22%
ffmpeg	221.90	62	62	0.01	99% 01%
scimark2	22.41	0.28	76	0.00	99% 01%
stream	7.00	0.35	33	0.01	99% 01%
ramspeed	2019.25	30.20	24	0.00	83% 17%
idle	1.09	0.01	99	0.60	48% 52%
phpbench	649.72	15.35	75	0.00	97% 03%
pybench	267.71	1.60	75	0.00	99% 01%
dbench	915.72	58.20	0	0.03	01% 99%
network-loopback	90.92	25.50	54	0.00	11% 89%

Table 2: Information on Phoronix Benchmarks

We have investigated benchmark stability over the 32 runs. Having reverse-engineered the Phoronix launching process and identified the trace parts about the 32 distinct runs, we evaluated the stability of the considered profile metrics (number of events, execution time, kernel and user time, number of cores). Considering the benchmark execution time, for example, we have computed the mean value, the maximum, the minimum and the standard deviation. The latter is close to zero meaning that the benchmarks are stable. The only exception is `idle` whose variation may be explained by its short execution time ($6ms$). The analysis of the other parameters shows similar results.

3.5 Benchmark Types

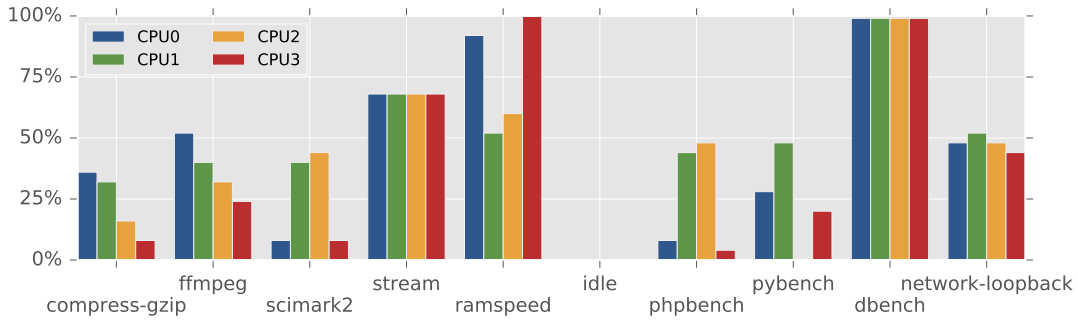
A first simple classification of Phoronix benchmark is to consider the ratio of kernel versus user operation. Table 2 gives this ratio and shows that there are only 4 benchmarks spending significant time in kernel mode. It is worth noting that the ratio here is computed over benchmarks' useful execution time and ignoring the idle time. For `idle`, for example, this represents only 1% of its total execution time. We can conclude that the others are either CPU- or memory intensive.

If we use the classification of kernel events introduced in Section 3.2 and use the number of traced events, we obtain the following kernel profiles (Figure 2b). We can clearly see that there is no benchmark testing the graphics subsystem (no graphics events) and that `network-loopback` and `dbench` respectively test the network and the disk. Indeed, they are the only ones with a significant amount of respectively *Network* and *Disk* events.

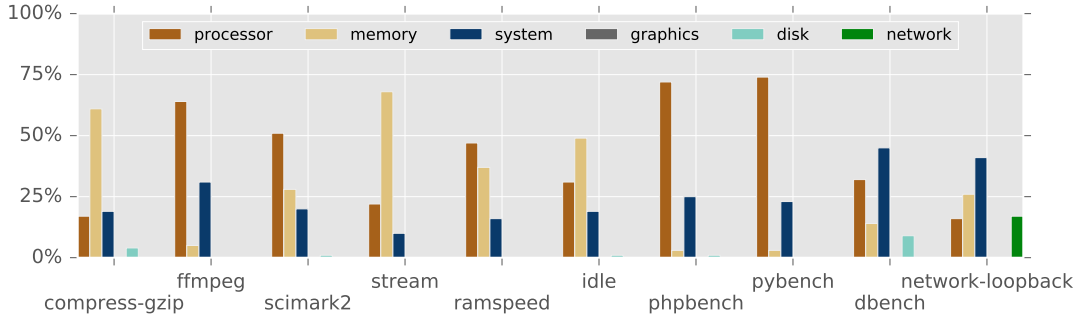
If we consider the benchmarks tagged as *Memory* within Phoronix, `stream` has an important kernel activity and its kernel profile confirms the frequent usage of memory-related functions. However, the profile of `scimark2` is different and to verify whether the benchmark is indeed memory-intensive one should consider its execution in user mode.

In the *Processor* family, `ffmpeg` and `scimark2` have the expected kernel profiles with predominant *Processor* events. `compress-gzip`, however, shows an important memory management activity so the profile computation should consider the user-level information.

Our analysis of the *System* Phoronix family made us understand that it includes various benchmarks testing different software systems (or layers, or middleware) and it does not necessarily focus on the operating system level. `idle` does test the operating system and quantifies the execution time of a program doing nothing. However, `phpbench` and `pybench`, which, by the



(a) CPU Usage and Parallelization.



(b) Kernel Operation

Figure 2: Phoronix Kernel Profiles

way, have similar kernel profiles, respectively test the performances of PHP and Python code.

3.6 CPU Usage and Parallelization

An interesting aspect we have investigated is the way benchmarks use the available processors. In the case of the UDOO platform, we can see is that benchmarks have quite different parallelization schemes (Figure 2a). The `idle` benchmark does not use the CPU, as expected.The `pybench` benchmark uses only 3 CPU out of 4. The other benchmarks do use the 4 processors but only `stream` and `network-loppback` benchmarks are totally balanced. `stream` uses the CPUs at 68%, while `network-loppback` use them around 45% if the time.

Another interesting observation is that there are two couples with quite similar CPU usage profiles. These are `scimark2` and `phpbench`, on one hand, and `compress-gzip` and `ffmpeg`, on the other. However `scimark2` and `phpbench` belong to the *Processor* and *System* family respectively. As for `compress-gzip` and `ffmpeg`, the two being of the *Processor* family, it may be better to consider the `ffmpeg` benchmark which runs longer but makes a more efficient usage of the platform processors.

3.7 Memory Usage Profile

Tracing the user-level memory functions provides interesting information about the differences between benchmarks. The maximum memory allocated by benchmarks, for example, is quite

Benchmark	Memory (KB)	Instructions repartition (Compute / Memory)
compress-gzip	138	72% / 28%
ffmpeg	5 210	71% / 29%
scimark2	16 779	64% / 36%
stream	9	58% / 42%
ramspeed	3 456 108	49% / 51%
idle	2	56% / 44%
phpbench	3 225	54% / 46%
pybench	1 827	56% / 44%
dbench	225 608	77% / 23%
network-loopback	1 123	55% / 45%

Table 3: Maximum Memory and Instructions repartition

varying (Table 3). Indeed, only `ramspeed` with $3.45GB$ uses almost all virtual memory available on the UDOO board (4GB). `stream` and `idle` are memory light-weight as they consume respectively $9KB$ and $2KB$.

The results from profiling of user-level operations and classifying them into computational and memory-related are given in the third row of Table 3. These confirm that the *Memory* benchmarks, `ramspeed` and `stream` do use intensively the memory as expected. The other benchmarks which reveal to be memory-intensive are the *System* ones we have selected, namely `phpbench` and `pybench`. As for `idle` and `network-loopback`, these are not representative as the time spend in user mode is very short (0.6% of the total execution time for `idle` and 5% for `network-loopback`).

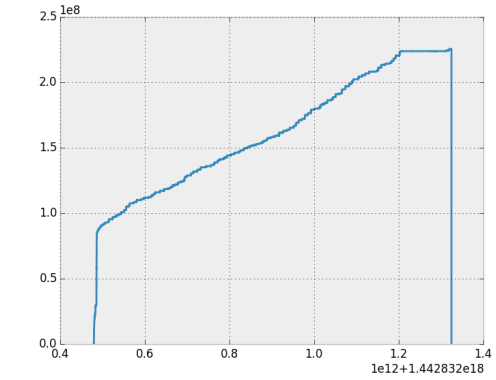
Figure 3 gives the evolution of the memory usage of benchmarks over time and shows that there are various behaviors.

4 Related Work

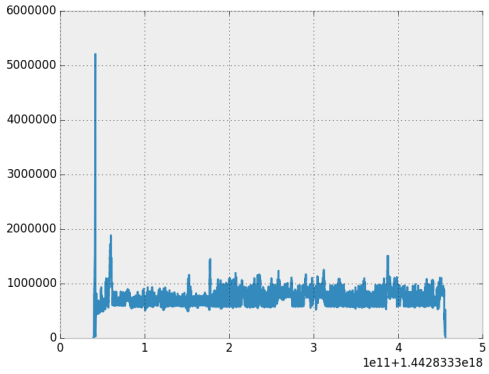
Current benchmark-oriented efforts [8, 18, 4] focus on the problems of providing a set of benchmarks which is to be *complete*, *portable* and *easy* to use. Indeed, the goal is to provide benchmarks that cover different performance aspects, support different platforms and can be automatically downloaded, installed and executed. However, the classification of benchmarks is *ad hoc* and there is no detailed information about their functionality. Our proposal is a step forward and allows for automatic profiling of benchmarks and providing the user with comprehensive basis for comparison.

Our proposal can be seen as a profiling tool for benchmarks. However, existing profiling tools [19, 20, 21, 22] typically provide detailed low-level information on a particular system aspect. Moreover, they are system dependent. Our proposal is applicable to all types of benchmarks, on different platforms and provides a macroscopic vision of their behavior.

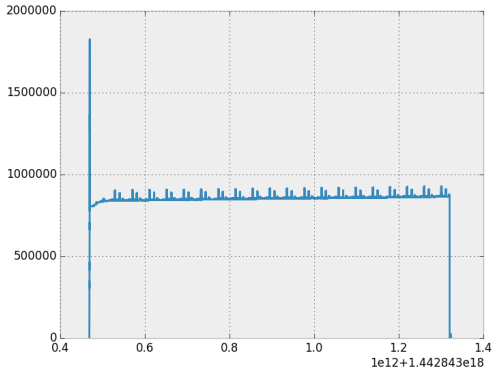
The major aspects of our profile computation are trace analysis and workflow management. Concerning trace analysis, most existing tools are system and format specific [23, 24, 25] and limit themselves to time-chart visualizations and basic statistics. In our work, trace analysis is brought to a higher level of abstraction. It is based on generic data representation of traces and thus may be applied to execution traces (hence benchmarks) from different systems. It is not organized as a set of predefined and thus limited treatments but may be configured and enriched to better respond to the user needs. Finally, its structuration in terms of a deterministic workflow allows for automation and reproducibility of the analysis process.



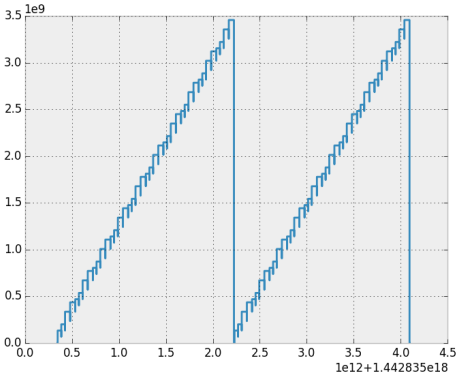
(a) dbench



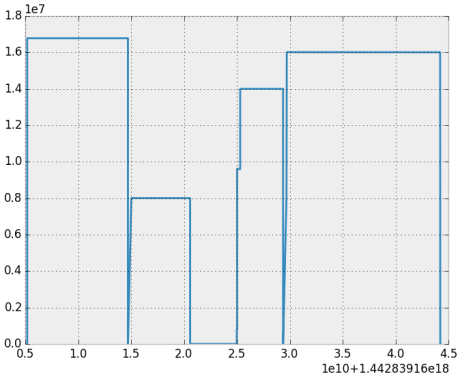
(b) ffmpeg



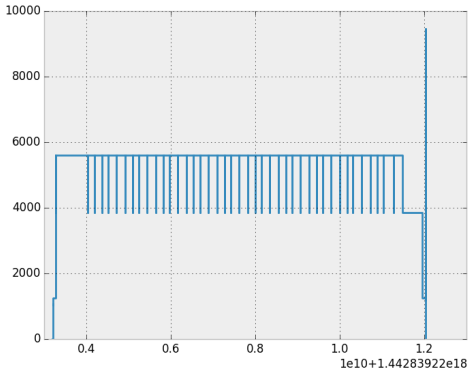
(c) pybench



(d) ramspeed



(e) scimark2



(f) stream

Figure 3: Memory usage (bytes) over time (ns).

As for workflow-oriented tools [26, 27], they focus on aspects like formal specification, automation, optimisation and reproducibility of computations. Generic trace analysis and especially the problem of huge traces have not been considered.

5 Conclusion and Ongoing Work

We have presented in this paper a workflow-based tool for automatic profiling of benchmarks. The result is a unified profile which characterizes a benchmark, allows the comparison among benchmarks and thus facilitates the choices of a system performance analyst. We have illustrated our approach with the Phoronix Test Suite and have experimented with embedded and desktop Linux-based platforms. We have successfully produced the profiles for several benchmarks exhibiting their different characteristics. Our experimentation puts forward the fact that the initially provided description is far from sufficient for understanding the way benchmarks test the target system.

In our work we have taken advantage of workflows' useful features such as automation, result caching and reproducibility. However, most workflow systems do not properly address the data management issues when it comes to manipulating big data sets. In this regard, we have shown that workflow tools should provide for pipelining, streaming and parallel computations. An ongoing collaboration with the VisTrails team brings these features to the VisTrails tool.

The benchmark profiles we provide are easy to understand and to compare. They are unified as they do not depend on the specifics of the benchmarks. However, in many cases it is the semantic specifics of the benchmark that makes its importance. A long term research objective would be to provide generic means for reflecting the benchmark specifics into the profile and thus help even more the performance evaluation work of an analysis.

References

[1] JoseBacelar Almeida, MariaJoao Frade, JorgeSousa Pinto, and Simao Melo de Sousa. An overview of formal methods tools and techniques. In *Rigorous Software Development*, Undergraduate Topics in Computer Science, pages 15–44. Springer London, 2011.

[2] Intel. Intel MPI Benchmarks 4.0 Update 2. `https://software.intel.com/en-us/articles/intel-mpi-benchmarks`.

[3] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[4] Future Benchmarks and Performance Tests. `http://www.futuremark.com/`.

[5] ApacheBench. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[6] HP. The httperf tool. `http://www.hpl.hp.com/research/linux/httperf/`.

[7] Linux Benchmark Suite Homepage. `http://lbs.sourceforge.net/`.

[8] Phoronix Test Suite. `http://www.phoronix-test-suite.com/`.

[9] *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, New York, NY, USA, 2000.

[10] Linux Trace Tool new generation. `http://lttng.org/`.

[11] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, 2009.

[12] Kuo-Yi Chen, Yuan-Hao Chang, Pei-Shu Liao, Pen-Chung Yew, Sheng-Wei Cheng, and Tien-Fu Chen. Selective Profiling for OS Scalability Study on Multicore Systems. *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 174–181, 2013.

[13] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos Scheidegger, Claudio T Silva, and Huy T Vo. VisTrails : Visualization meets Data Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 745–747, 2006.

[14] Generoso Pagano and al. Trace Management and Analysis for Embedded Systems. *IEEE 7th International Symposium on Embedded Multicore Socs*, September 2013.

[15] R Jain. *The Art Of Computer Systems Performance Analysis*. 1991.

[16] Udoo. `http://www.udoo.org`.

[17] Juno. `http://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php`.

[18] Industry-Standard Benchmarks for Embedded Systems. `http://www.eembc.org/benchmark/products.php/`.

[19] M. Yamamoto, M. Ono, K. Nakashima, and A. Hirai. Unified performance profiling of an entire virtualized environment. In *Computing and Networking (CANDAR), 2014 Second International Symposium on*, pages 106–115, Dec 2014.

[20] Jean-Philippe Gouigoux. *Practical Performance Profiling: Improving the Efficiency of .NET Code*. Red gate books, United Kingdom, 2012.

[21] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008.

[22] Tomislav Janjusic and Christos Kartsaklis. Glprof: A gprof inspired, callgraph-oriented per-object disseminating memory access multi-cache profiler. *Procedia Computer Science*, 51:1363 – 1372, 2015. International Conference On Computational Science, {ICCS} 2015Computational Science at the Gates of Nature.

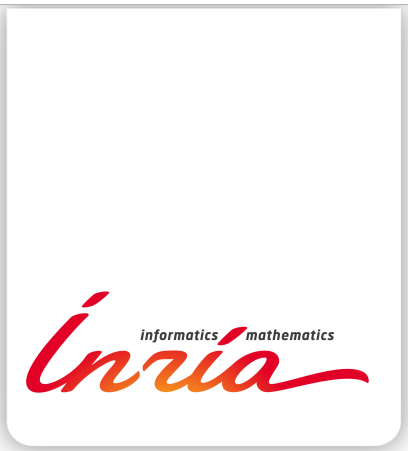
[23] Andreas Knüpfer and al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011, ZIH, Dresden, September 2011*, pages 79–91, 2011.

[24] David Couturier and Michel R. Dagenais. Lttng CLUST: A system-wide unified CPU and GPU tracing tool for opencl applications. *Adv. Software Engineering*, 2015:940628:1–940628:14, 2015.

[25] Carlos Prada-Rojas, Miguel Santana, Serge De-Paoli, and Xavier Raynaud. Summarizing Embedded Execution Traces through a Compact View. In *Conference on System Software, SoC and Silicon Debug S4D*, 2010.

[26] Sarah Cohen-Boulakia and Ulf Leser. Search, adapt and reuse: The future of scientific workflows. *SIGMOD Records*, 40(2):1187–1189, 2011.

[27] Jun Qin and Thomas Fahringer. *Scientific Workflows, Programming, Optimization, and Synthesis with ASKALON and AWDL*, volume ISBN 978-3-642-30714-0. Springer, 2012.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399